# Grammatical Characterizations of P and PSPACE

AUTHOR(S):

CHEN, Zhi-Zhong; TODA, Seinosuke

CITATION:

CHEN, Zhi-Zhong ...[et al]. Grammatical Characterizations of P and PSPACE. 数理解析研究所講究録 1990, 731: 226-237

ISSUE DATE:

1990-10

URL:

http://hdl.handle.net/2433/101963

数理解析研究所講究録 226
第 731 巻 1990 年 226-237

# Grammatical Characterizations of P and PSPACE

Zhi-Zhong CHEN and Seinosuke TODA

Department of Computer Science,
Univ. of Electro-Communications,
Chofu-shi, Tokyo 182.

**SUMMARY** The notion of alternating context-free grammar (ACFG for short) was introduced by Moriya in 1989. In this paper, we study the relationships between some complexity classes and the classes of languages generated by restricted types of ACFG's. Two restricted types of ACFG's considered are linear ACFG's and $\varepsilon$-free ACFG's. For an ACFG $G$, let $L_{left}(G)$ denote the language of terminal strings generated by leftmost derivations in $G$. Let $\mathbf{ACFL}^{left}_{\varepsilon-free} = \{L_{left}(G) \mid G$ is an $\varepsilon$-free ACFG $\}$ and $\mathbf{ACFL}_{linear} = \{L_{left}(G) \mid G$ is a linear ACFG's $\}$. The main results of the present paper are as follows:

(1) the class of languages that are *log*-space many-one reducible to languages in $\mathbf{ACFL}_{linear}$ is equivalent to **P**, and

(2) the class of languages that are *log*-space many-one reducible to languages in $\mathbf{ACFL}^{left}_{\varepsilon-free}$ is equivalent to **PSPACE**.

## 1 Introduction

Alternating context-free grammars (ACFG for short) were introduced by Moriya[6] as an interesting generalization of context-free grammars (CFG for short). Moriya also investigated some elementary properties of ACFG's and the relationships among many classes defined by ACFG's. In this paper, we investigate the relationships between some complexity classes and some restricted types of ACFG's. Two restricted types of ACFG's are considered in this paper. They are $\varepsilon$-free ACFG's and linear ACFG's. Intuitively speaking, $\varepsilon$-free ACFG's (resp., linear ACFG's) are an alternating analogue to $\varepsilon$-free CFG's (resp., linear ACFG's). The formal definitions of them are given in Section 2.

The relationships between grammars and complexity classes have been well studied by many researchers[2],[3],[9],[10]. Especially, Sudborough showed that $\mathbf{NL}=\mathbf{LOG}(\mathbf{CFL}_{linear})$[10]

and aux$\mathbf{PDA}(poly)$=$\mathbf{LOG(CFL)}$[9], where $\mathbf{NL}$ and aux$\mathbf{PDA}(poly)$[7] are the classes of languages accepted by nondeterministic Turing machines in logarithmic space and auxliary pushdown automata in polynomial time, respectively, and $\mathbf{LOG}(\mathbf{CFL}_{linear})$ and $\mathbf{LOG(CFL)}$ are the classes of languages *log*-space many-one reducible to linear context-free languages and (unrestricted) context-free languages, respectively. These results show the computational complexity of recognizing context-free languages and, simultaneously, they give grammatical charaterizations for some interesting complexity classes. In this paper, we consider the computational complexity of recognizing some restricted types of alternating context-free languages. Of particular interest is to find some grammatical charaterizations for some other complexity classes. One of our main results is that $\mathbf{P}$=$\mathbf{LOG}(\mathbf{ACFL}_{linear})$, where $\mathbf{P}$ is the class of languages accepted by deterministic Turing machines in polynomial time and $\mathbf{LOG}(\mathbf{ACFL}_{linear})$ is the class of languages that is *log*-space many-one reducible to those generated by linear ACFG's. Noting that $\mathbf{P}$=$\mathbf{ASPACE}(log)$[1] (the class of languages accepted by alternating Turing machines in logarithmic space), the result can be viewed as an alternating counterpart of Sudborough's fist result. In other words, the notion of alternation in a grammatical sense works for a similar meaning to the case of Turing machine. We also investigate the relationship between $\varepsilon$-free ACFG's and $\mathbf{PSPACE}$ (the class of languages accepted by deterministic Turing machines in polynomial space). In this paper, derivations in each $\varepsilon$-free ACFG are restricted to be leftmost and the language generated by each $\varepsilon$-free ACFG is the language of terminal strings generated by leftmost derivations. The reason is that there exists an $\varepsilon$-free ACFG in which some terminal strings are generated in that grammar but are not able to be generated by any leftmost derivations in that grammar. Hence, the class of languages generated by $\varepsilon$-free ACFG's would be different from the class of those generated by leftmost derivations of $\varepsilon$-free ACFG's. In this restricted sense, we prove that $\mathbf{PSPACE}$ is equivalent to the class of languages *log*-space many-one reducible to languages generated by $\varepsilon$-free ACFG's. We note that our proof technique does not work for the class of $\varepsilon$-free alternating context-free languages (in a usual sense). It is currently unknown even whether the class of $\varepsilon$-free alternating context-free languages is computationally equivalent to the class of those restricted in the above sense. It is an interesting open question.

The present paper is organized as follows. In the next section, we define some notions and notations. In particular, we define a variation of alternating pushdown automata for the sake of simplifying the proof. In Section 3, we show the relationship between linear ACFL's and

**P**. In Section 4, we show the relationship between $\varepsilon$-free ACFL's and **PSPACE**. In the final section, we exhibit some interesting open questions.

## 2 Preliminaries

The reader is assumed to be familiar with the basic concepts in formal language and computational complexity theories. Unless stated otherwise, basic notations in this paper follow Hopcroft and Ullman[4]. Below, by $\varepsilon$, $|w|$, and $\#Q$, we denote the empty string, the length of string $w$, and the cardinality of a finite set $Q$, respectively.

### 2.1 Alternating context-free grammars

The following fundamental definition is cited from Moriya's paper[6].

**[Definition 2.1]** An *alternating context-free grammar* (ACFG for short) is a quintuple $G = (N, U, \Sigma, P, S)$, where $(N, \Sigma, P, S)$ is a context-free grammar (CFG for short), called the *underlying* CFG of $G$, and $U$ is a subset of $N$. Elements of $U$ and $N - U$ are called *universal* and *existential nonterminals*, respectively. A production whose lefthand side is an existential (universal) nonterminal is called an *existential* (resp., *universal*) *production.*

Let $G = (N, U, \Sigma, P, S)$ be an ACFG and $\alpha$ be in $(N \cup \Sigma)^*$. A finite tree $T$ is called a *leftmost derivation* for $G$ from $\alpha$ if the following properties are satisfied:

(a) Each node $\tau$ is labeled with a string in $(N \cup \Sigma)^*$, denoted $\ell(\tau)$; in particular, the root of $T$ is labeled with $\alpha$.

(b) If $\tau$ is an internal node of $T$ such that $\ell(\tau) = xA\beta$ with $x \in \Sigma^*$, $A \in N - U$ and $A \to \gamma$ is a production in $P$, then $\tau$ has exactly one son $\tau'$ labeled with $\ell(\tau') = x\gamma\beta$. In this case, $\tau$ is called an *existential node.*

(c) If $\tau$ is an internal node of $T$ such that $\ell(\tau) = xA\beta$ with $x \in \Sigma^*$, $A \in U$ and $A \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_k$ are the $A$-productions (i.e. productions whose lefthand side is $A$) in $P$, then $\tau$ has exactly $k$ sons $\tau'_1, \tau'_2, \cdots, \tau'_k$ with $\ell(\tau'_i) = x\gamma_i\beta$, $1 \leq i \leq k$. In this case, $\tau$ is called a *universal node.*

For each node $\tau$ of a leftmost derivation $T$, the *value* of $\tau$, $val(\tau)$, is inductively defined as

follows.

$$val(\tau) = \begin{cases} \ell(\tau) & \text{if } \tau \text{ is a leaf of } T, \\ val(\rho) & \text{if } \tau \text{ is an existential node and } \rho \text{ is its son,} \\ w & \text{if } \tau \text{ is a universal node and} \\ & \quad w = val(\rho) \text{ for each son } \rho \text{ of } \tau, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The *value of a leftmost derivation* $T$, denoted $value(T)$, is defined to be the value of its root. Notice that the value of a leftmost derivation may be undefined. For a leftmost derivation $T$, if its value is defined, then it is said to be *valid*, and then it is said to generate $value(T)$ from the label of the root. Note that all the leaves of a valid leftmost derivation $T$ have the same label, $value(T)$. The label of a node of a leftmost derivation whose root is labeled with the start symbol is called a *sentential form.* A *terminating leftmost derivation* is a leftmost derivation whose leaves each have a label in $\Sigma^*$. An *acceptable leftmost derivation* is a terminating valid leftmost derivation whose root is labeled with the start symbol. The *language generated* by $G$ is defined to be the set

$$L_{left}(G) = \{value(T) \mid T \text{ is an acceptable leftmost derivation in } G\}$$

A language $L$ is an *alternating context-free language* (ACFL for short) if $L = L_{left}(G)$ for some ACFG $G$.

ACFG's can be classified according to the form of their productions. In this paper, we consider two restricted types of ACFG's which will be defined as follows. Let $G = (N, U, \Sigma, P, S)$ be an ACFG.

**[Definition 2.2]** $G$ is *linear* if every production is of the form $A \to xBy$ or $A \to x$ for some $x, y \in \Sigma^*$ and $B \in N$. An ACFL is *linear* if it can be generated by some linear ACFG.

**[Definition 2.3]** A production is called an *$\varepsilon$-production* if it is of the form $A \to \varepsilon$ for some $A \in N$. We say that $G$ is *$\varepsilon$-free* if it has no $\varepsilon$-productions. An ACFL is said to be $\varepsilon$-free if it can be generated by some $\varepsilon$-free ACFG.

Notice that all derivations in any linear ACFG are obviously leftmost. Hence, our result in Section 3 has no restriction while the result in Section 4 has a restriction.

### 2.2 Complexity classes and notations

The main goal of this paper is to find the relations between ACFL's and complexity classes. As usual[4], we denote by **PSPACE** and **P** the classes of languages accepted by deterministic

Turing machines in polynomial space and in polynomial time, respectively. We denote by **ATIME**(*poly*) and **ASPACE**(*log*) the classes of languages accepted by alternating Turing machines in polynomial time and in logarithmic space, respectively. A language $L'$ is *log-space many-one reducible* to a language $L$ ($L'$ is $\leq_m^{log}$-reducible to $L$) if there exists a *log*-space computable function $f$ such that for every $x$, $x \in L'$ iff $f(x) \in L$. We write $L' \leq_m^{log} L$ if $L'$ is $\leq_m^{log}$-reducible to $L$. For a class **C** of languages, we say that a language $L$ is **C**-complete if $L \in$**C** and every language in **C** is $\leq_m^{log}$-reducible to $L$. We say that a class **C** is *closed under* $\leq_m^{log}$*-reducibility* if $L' \leq_m^{log} L$ and $L \in$**C** implies $L' \in$**C** for all languages $L'$ and $L$. For a language $L$, we define $\mathbf{LOG}(L) = \{L' \mid L' \leq_m^{log} L\}$. Furthermore, for a class **C** of languages, we define $\mathbf{LOG}(\mathbf{C}) = \cup_{L \in \mathbf{C}} \mathbf{LOG}(L)$.

We denote by $\mathbf{ACFL}_{\varepsilon-free}^{left}$ and $\mathbf{ACFL}_{linear}$ the classes of ACFL's, $\varepsilon$-free ACFL's, and linear ACFL's, respectively.

### 2.3 State-free alternating pushdown automata

In this section, we introduce a variation of alternating pushdown automata for the sake of simplifying the proofs of our main results.

[**Definition 2.4**] A *state-free alternating pushdown automaton* (SF_APDA for short) $M$ is a five-tuple $M = (\Sigma, \Gamma, \Pi, \delta, Z_0)$, where $\Sigma$ is a finite *input alphabet*, $\Gamma$ is a finite alphabet of *pushdown stack symbols*, $\Pi$ is a subset of $\Gamma$, $Z_0 \in \Gamma$ is a particular pushdown stack symbol that appears initially on the pushdown stack, and $\delta$ is a transition function from $(\Sigma \cup \{\varepsilon\}) \times \Gamma$ to the finite subsets of $\Gamma^*$. Each element in $\Gamma - \Pi$ (resp., $\Pi$) is called a *existential* (resp., *universal*) pushdown stack symbol.

An *instantaneous description(ID)* of $M$ has the form $(w, \gamma)$, where

(1) $w$ represents the unused portion of the input. At this point, the input head reads the leftmost symbol of $w$.

(2) $\gamma$ represents the content of the pushdown stack. The leftmost symbol of $\gamma$ is the topmost pushdown symbol. If $\gamma = \varepsilon$, then pushdown stack is assumed to be empty.

The *initial ID* of $M$ on input $w$ is $(w, Z_0)$; The *accepting ID* is $(\varepsilon, \varepsilon)$. A *computation tree* of $M$ on input $w$ is a finite rooted tree defined as follows:

**(a)** Each node $\tau$ is labeled with an $ID$ of $M$, denoted $\ell(\tau)$; in particular, the root is labeled with the initial $ID$ of $M$ on input $w$.

**(b)** If $\tau$ is an internal node such that $\ell(\tau) = (ax, Z\alpha)$ with $a \in \Sigma \cup \{\varepsilon\}$ and $Z \in \Gamma - \Pi$ and if $\beta \in \delta(a, Z) \cup \delta(\varepsilon, Z)$, then $\tau$ has exactly one son $\tau'$ such that $\ell(\tau') = (x, \beta\alpha)$ if $\beta \in \delta(a, Z)$ and $\ell(\tau') = (ax, \beta\alpha)$ if $\beta \in \delta(\varepsilon, Z)$.

**(c)** If $\tau$ is an internal node such that $\ell(\tau) = (ax, Z\alpha)$ with $a \in \Sigma \cup \{\varepsilon\}$ and $Z \in \Pi$ and if $\delta(a, Z) \cup \delta(\varepsilon, Z) = \{\beta_i \mid 1 \leq i \leq k\}$, then $\tau$ has exactly $k$ sons $\tau_1, \tau_2, \cdots, \tau_k$ such that for $1 \leq i \leq k$, $\ell(\tau_i) = (x, \beta_i\alpha)$ if $\beta_i \in \delta(a, Z)$ and $\ell(\tau_i) = (ax, \beta_i\alpha)$ if $\beta_i \in \delta(\varepsilon, Z)$.

An *accepting computation tree* of $M$ on input $w$ is a computation tree of $M$ on input $w$, whose leaves each are labeled with the accepting $ID$. We say that $M$ *accepts* $w$ if there is an accepting computation tree of $M$ on input $w$. The language of input strings accepted by $M$ is denoted by $L(M)$.

We say that a move from $(u, Z\alpha)$ to $(v, \beta)$ is a *$\varepsilon$-pop-move* if $v = u$ and $\beta = \alpha$ (i.e. in this move, the machine pops up the top symbol of the pushdown store without moving the input head).

## 3 A characterization of PSPACE in terms of $\varepsilon$-free ACFG's

In this section, we characterize **PSPACE** in terms of $\varepsilon$-free ACFL's. We first show that all languages in $\mathbf{ACFL}^{left}_{\varepsilon-free}$ are in **PSPACE**, and then show that a **PSPACE**-complete language is $\leq^{log}_{m}$-reducible to a language in $\mathbf{ACFL}^{left}_{\varepsilon-free}$.

**[Lemma 3.1]** $\mathbf{ACFL}^{left}_{\varepsilon-free} \subseteq \mathbf{PSPACE}$.

(Proof) Let $L = L_{left}(G)$, where $G = (N, U, \Sigma, P, S)$ is an $\varepsilon$-free ACFG. It has been shown that $\mathbf{ATIME}(poly) = \mathbf{PSPACE}$[1]. So we prove this lemma by constructing an alternating Turing machine $M$ which accepts $L$ in polynomial time.

$M$ simply simulates leftmost derivations for $G$ from $S$ by remembering a sentential form on its work tape. More precisely, given an input $w$ with length $n$, $M$ starts its computation with writing the start symbol $S$ on its worktape. At some time, suppose $M$ has a string $xB\beta$ on its work tape, where $x \in \Sigma^*$, $B \in N$, and $\beta \in N^*$. Then, if $B \in N - U$, then $M$ existentially chooses a $B$-production and replaces $B$ by its righthand side. If $B \in U$, then $M$ universally chooses a $B$-production and replaces $B$ by its righthand side. $M$ operates along this way until the string on the work tape becomes a string $u \in \Sigma^*$. At the last time, if $u = w$, then $M$ enters an accepting state; otherwise, $M$ enters a rejecting state. Furthermore, at any moment of its computation, if the length of the string on the work tape exceeds $n$, then $M$ enters a

rejecting state immediately.

It is easy to see that $M$ accepts $L_{left}(G)$. To see that $M$ is polynomial time bounded, we note that at least $\#P$ applications of productions must increase the length of the sentential form by at least 1 or increase the number of terminals in the sentential form by at least 1. We also note that only the number of terminals can be increased once the sentential form changes to another one with length $|w|$. From these, we know that at most $2 \cdot \#P \cdot |w|$ applications of productions is enough to generate $w$. Thus $M$ is polynomial time bounded. □

In order to show **PSPACE**$\subseteq$**LOG**(**ACFL**$^{left}_{\varepsilon-free}$), we use a restricted type of SF_APDA's as a tool. An SF_APDA is *$\varepsilon$-pop-free* if it does no $\varepsilon$-pop-moves on any input. We denote by **SF_APDA**$_{\varepsilon-pop-free}$ the class of languages accepted by $\varepsilon$-pop-free SF_APDA's.

The following lemma will be used in the proof of the main result of this section.

[**Lemma 3.2**] **SF_APDA**$_{\varepsilon-pop-free} \subseteq$ **ACFL**$^{left}_{\varepsilon-free}$.

Let **3QBF** denote the set of true quantified Boolean formulas

$$F = (q_1 x_1)(q_2 x_2) \cdots (q_n x_n) E$$

where $q_i \in \{\forall, \exists\}$ and $E$ is a conjunction of 3-literal disjunctive clauses.

[**Proposition 3.3**] **3QBF** is $\leq^{log}_m$-complete for **PSPACE**[(8),(11)].

[**Theorem 3.4**] **LOG**(**ACFL**$^{left}_{\varepsilon-free}$) =**PSPACE**.

(Proof) ($\subseteq$) This follows from Lemma 3.1 and the fact that **PSPACE** is closed under $\leq^{log}_m$.

($\supseteq$) From Lemma 3.2 and Proposition 3.3, it suffices to show that **3QBF** is $\leq^{log}_m$-reducible to a language accepted by an $\varepsilon$-pop-free SF_APDA. Let $F = (q_1 x_1)(q_2 x_2) \cdots (q_n x_n) C_1 C_2 \cdots C_n$ be a quantified boolean formula, where $C_j = (l^{(j)}_1 + l^{(j)}_2 + l^{(j)}_3)$ and each $l^{(j)}_i$ is either a variable (i.e. a positive literal) or a negation of a variable (i.e. a negative literal). We first encode $F$ into a string $\tilde{F}$ in $\{\not\in, \exists, \forall, a, (, ), \flat, +, \$\}^+$ as follows:

$$\tilde{F} = q_1 a^1 q_2 a^2 \cdots q_m a^m \tilde{C}_1 \tilde{C}_2 \cdots \tilde{C}_n \not\in^{m^2} \$,$$

where

**(1)** $q_i \in \{\exists, \forall\}$,

**(2)** $\tilde{C}_j = \flat^{m^2}(\tilde{l}^{(j)}_1 + \tilde{l}^{(j)}_2 + \tilde{l}^{(j)}_3)$, and

**(3)** $\tilde{l}^{(j)}_i = a^k 1$ if $l^{(j)}_i$ is a positive literal $x_k$, and $\tilde{l}^{(j)}_i = a^k 0$ if $l^{(j)}_i$ is a negative literal $\bar{x}_k$.

We construct an $\varepsilon$-pop-free SF_APDA $M$ working on an input string from $\{\not\in, \exists, \forall, a, (, ), \flat, +,$ Intuitively speaking, $M$ operates as follows. At the beginning of its computation, it universally chooses one of two actions. One action is to check whether the current input is in the set $E$ defined by a regular expression

$$(\{\exists,\forall\}\{a\}^+)^+(\{\flat\}^+\{(\}\{a\}^+\{0,1\}\{+\}\{a\}^+\{0,1\}\{+\}\{a\}^+\{0,1\}\{)\})^+\{\not\in\}^+\{\$\}.$$

Notice that $\tilde{F}$ above is in $E$. If the input is in $E$, then $M$ accepts the input; otherwise, it rejects the input. The other action is to check whether the current input is (an encoding string of) a true quantified boolean formula. This check is done in the following manner. M first chooses an assignment to each variable either existentially or universally according to the quantifier bounding this variable and it keeps these assignments with variable names by the pushdown stack. After that, it universally chooses one of clauses and checks whether the clause is true on the guessed assignment. In order to do this check, $M$ existentially chooses one of variable names with associated assignment from the pushdown store and further, it existentially chooses one of literals in the cluase. After that, $M$ compares both with each other. If both agree with each other (i.e. the cluase is true), then $M$ accepts the input; otherwise, it rejects.

We notice that $M$ has to move the input head when poping up a pushdown symbol. Below, the symbols $\flat^{m^2}$ in $\tilde{F}$ above will be used for guessing one of variable names on the pushdown store. Furthermore, the symbols $\not\in^{m^2}\$$ will be used for making pushdown store empty. □

## 4 A charaterization of P in terms of linear ACFG's

In this section, we charaterize **P** in terms of linear ACFG's. We first show that all languages in $\mathbf{ACFL}_{linear}$ are in **P** and then show that a **P**-complete language is $\leq_m^{log}$-reducible to a language in $\mathbf{ACFL}_{linear}$.

**[Lemma 4.1]** $\mathbf{ACFL}_{linear} \subseteq \mathbf{P}$.

(Proof) Let $L = L_{left}(G)$, where $G = (N, U, \Sigma, P, S)$ is a linear ACFG. Since it has been shown in [1] that $\mathbf{ASPACE}(log)=\mathbf{P}$, we prove this lemma by constructing an alternating Turing machine $M$ which accepts $L$ in logarithmic space.

$M$ simply simulates leftmost derivations for $G$ from $S$ by remembering a nonterminal and two positions $p_1$, $p_2$ on the current input. Given an input $w$ with length $n$, $M$ starts its computation with writing the start symbol $S$ on the work tape and setting $p_1 = 1$ and $p_2 = n$.

At some time, suppose that $M$ has a nonterminal $B$ on the work tape, If $B \in N - U$ ($B \in U$), then $M$ existentially (resp., universally) chooses a $B$-production, say $B \to xAy$ for $A \in N$, and replace $B$ by $A$. Furthermore, $M$ checks the following conditions: (1) the part of $w$ from the $p_1$'th symbol to the $p_1 + |x| - 1$'th symbol is $x$, (2) the part of $w$ from the $p_2 - |y| + 1$'th symbol to the $p_2$'th symbol is $y$. If $M$ does not succeed in this checking, then it enters a rejecting state immediately; otherwise, it increases $p_1$ by $|x|$ and decreases $p_2$ by $|y|$. $M$ operates along this manner until it chooses a production $A \to x$ for some $x \in \Sigma^*$ is chosen. At the last time, $M$ checks whether the part of $w$ from the $p_1$'th symbol to the $p_2$'th symbol is $x$. If so, $M$ enters an accepting state; otherwise, $M$ enters a rejecting state. Furthermore, at any moment of its computation, if the value of $p_1$ becomes larger than that of $p_2$, then $M$ enters a rejecting state immediately.

It is easy to see that $M$ accepts $L_{left}(G)$ and is log space bounded. □

In order to show that $\mathbf{P} \subseteq \mathbf{LOG(ACFL}_{linear}\mathbf{)}$, we use another restricted type of SF_APDA's as a technical tool.

[**Definition 4.1**] An SF_APDA is 1-*turn* if, in each move, it operates deterministically, pops up a pushdown symbol, and moves its input head after the time that it pops up a symbol from the pushdown stack. We denote by $\mathbf{SF\_APDA}_{1-turn}$ the class of languages accepted by 1-turn SF_APDA's.

[**Lemma 4.2**] $\mathbf{SF\_APDA}_{1-turn} \subseteq \mathbf{ACFL}_{linear}$.

In order to prove our next theorem, some more definitions are necessary.

An *n-node acyclic and/or graph* is a pair $G = (f, g)$, where

**(1)** $f$ is a function from $\{1, 2, \cdots, n\}$ to $\{\vee, \wedge\}$,

**(2)** $g$ is a function from $\{1, 2, \cdots, n\}$ to $2^{\{1,2,\cdots,n\}}$, and

**(3)** for every $i \in \{1, 2, \cdots, n\}$, $\sharp g(i) = 2$ or $0$, $j > i$ if $j \in g(i)$, and for every $j > i$, $g(j) = 0$ if $g(i) = 0$.

Let $G$ be a $n$-node acyclic and/or graph. For each $i \in \{1, 2, \cdots, n\}$, its *value* $\sigma(i)$ is defined inductively as follows.

$$\sigma(i) = \begin{cases} true & \text{if } g(i) = \emptyset \text{ and } f(i) = \vee, \\ false & \text{if } g(i) = \emptyset \text{ and } f(i) = \wedge, \\ \sigma(i) f(i) \sigma(k) & \text{if } g(i) = \{j, k\} \end{cases}$$

We define the value of $G$ to be $\sigma(1)$. Let

**AGAP**= $\{G \mid G$ is a $n$-node acyclic and/or graph whose value is *true*$\}$.

[**Proposition 4.3**] **AGAP** is $\leq_m^{log}$-complete for $\mathbf{P}^{(5)}$.

[**Theorem 4.4**] **LOG(ACFL**$_{linear}$**)=P**.

(Proof) ($\subseteq$) This follows from Lemma 4.1 and the fact that $P$ is closed under $\leq_m^{log}$.

($\supseteq$) From Lemma 4.2 and Proposition 4.3, it suffices to show that **AGAP** is $\leq_m^{log}$-reducible to a language accepted by a 1-turn SF_APDA.

Let $G = (f, g)$ be a $n$-node acyclic and/or graph and $m$ is the number of nodes $i$ in $G$ satisfying $g(i) \neq \emptyset$. We first encode $G$ into a string $\tilde{G}$ in $\{\wedge, \vee, a, b, \#, \not\in, \$\}^+$ as follows.

$$\tilde{G} = \#w_1\#w_2\# \cdots \#w_m\$\bar{w}_m \cdots \bar{w}_2\bar{w}_1\$$$

where

**(1)** $\bar{w}_i = ba^iba^i$ for $1 \leq i \leq m$,

**(2)** if $g(i) = \{j, k\}$, then $w_i = a^if(i)u_j \not\in u_k$ for $1 \leq i \leq m$, and

**(3)** for $1 \leq i \leq n$,

$$u_i = \begin{cases} a^ib & \text{if } \#g(i) = 2, \\ b & \text{if } \#g(j) = 0 \text{ and } f(i) = \vee, \\ bb & \text{if } \#g(i) = 0 \text{ and } f(i) = \wedge. \end{cases}$$

Below, for convenience, let $w_i$ be called the *information block* of node $i$ and let $\bar{w}_i$ be called the *match block* of node $i$ ($1 \leq i \leq m$).

We construct a 1-trun SF_APDA $M$ working on an input string from $\{\wedge, \vee, a, b, \#, \not\in, \$\}^*$. Intuitively speaking, $M$ operates as follows. At the beginning of its computation, it universally chooses one of two actions. One action is to check whether the current input is in the set $E$ defined by a regular expression

$$(\{\#\}\{a\}^+\{\vee, \wedge\}\{a\}^*\{b\}\{\varepsilon, b\}\{\not\in\}\{a\}^*\{b\}\{\varepsilon, b\})^+\{\$\}(\{b\}\{a\}^+)^+\{\$\}.$$

Notice that $\tilde{G}$ above is in $E$. If the input is in $E$, then $M$ accepts the input; otherwise, it rejects the input. The other action is to check whether the current input is (an encoding string of) a $n$-node acyclic and/or graph whose value is *true*. $M$ starts this checking by deterministically pushing the first node 1 onto its pushdown stack (hereafter, when we say "push a node $i$ onto the pushdown stack", it means that "push the name $a^ib$ of node $i$ onto the pushdown stack"). To check whether the value of node 1 is *true*, $M$ first finds the two sons

of node 1 from the information block of node 1 and then computes their values. $M$ finds the position of the information block of node 1 in $\tilde{G}$ existentially shifting the input head to some information block, say the information block of node $\ell_1$ ($\ell_1$ is desired to be 1). If $f(\ell_1) = \vee$ ($\wedge$), then $M$ existentially (resp., universally) chooses one son, say $j_1$ to see whether its value is *true*. If $g(j_1) \neq \emptyset$, then $M$ first puts $\ell_1 j_1$ onto its pushdown stack and then begins to check whether the value of node $j_1$ is *true*. This checking is similar to that for node 1. $M$ operates along this way until some node $j_k$ with $g(j_k) = \emptyset$ is chosen. If $f(j_k) = \wedge$, then $M$ rejects. If $f(j_k) = \vee$, then $M$ first puts $\ell_k$ onto its pushdown stack and then begins to check whether the current chosen nodes remembered on the pushdown stack consist of a real path in $G$. We next explain how to do this checking. Suppose that the list of (names of) nodes on the pushdown stack is

$$ba^{\ell_k} ba^{j_{k-1}} ba^{\ell_{k-1}} \cdots ba^{j_2} ba^{\ell_2} ba^{j_1} ba^{\ell_1} ba^1. \qquad (*)$$

Information blocks insure that each $j_i$ is really $\ell_i$'s son. What remains to check is whether $1 = \ell_1$ and $j_i = \ell_{i+1}$ $(1 \leq i \leq k-1)$. This check is done by using match blocks in $\tilde{G}$. We notice that $M$ has to be 1-turn. To this end, we construct $M$ so that it remembers the following instead of $(*)$ above

$$ba^{\ell_k} \#^* ba^{j_{k-1}} \#^* ba^{\ell_{k-1}} \#^* \cdots ba^{j_2} \#^* ba^{\ell_2} \#^* ba^{j_1} \#^* ba^{\ell_1} \#^* ba^1.$$

The length of each $\#^*$ is existentially chosen by $M$ and is desired to be the distance between the match block of $\ell_i$ and the block of $\ell_{i-1}$ $(2 \leq i \leq k)$ in $\tilde{G}$. $M$ deterministically skips useless match blocks in $\tilde{G}$ by popping $\#^*$'s. □

## 5 Concluding remarks

In summary, we have shown that $\mathbf{LOG}(\mathbf{ACFL}^{left}_{\varepsilon-free}) = \mathbf{PSPACE}$ and $\mathbf{LOG}(\mathbf{ACFL}_{linear}) = \mathbf{P}$, given new characterizations of **PSPACE** and **P**. Since it is possible that some derivation in some ACFG does not have a leftmost version, we have restricted to consider only leftmost derivations in the case of $\varepsilon$-free ACFG's. However, it is still possible that for every ACFG $G$, there is another ACFG $G'$ such that every derivation in $G'$ has a leftmost version. We have not been able to settle this question. We believe that a solution for this question will make the notion of ACFG more interesting.

### References

(1) A. K. Chandra, D. C. Kozen and L. J. Stockmeyer: "Alternation", J. ACM, 28, 1, pp.114-133(1981).

(2) S. A. Cook: "Charaterizations of pushdown machines in terms of time-bounded computers", J. ACM, 18, pp.4-18(1971).

(3) J. Engelfriet: "The complexity of languages generated by attribute grammars", SIAM J. Comput., 15, 1, pp.70-86(1986).

(4) J. E. Hopcroft and J. D. Ullman: "Introduction to automata theory, languages, and computation", Addision-Wesley, Reading, Mass.(1979).

(5) N. Immerman: "Number of quantifiers is better than number of tape cells", J. Comput. Syst. Sci., 22, pp.384-406(1981).

(6) E. Moriya: "A grammatical charaterization of alternating pushdown automata", Theoret. Comput. Sci., 67, pp. 75-85(1989).

(7) W. L. Ruzzo: "Tree-bounded alternation", J. Comput. Syst. Sci., 21, pp.218-235(1980).

(8) L. J. Stockmeyer: "The polynomial-time hierarchy", Theoret. Comput. Sci., 3, pp.1-22(1977).

(9) I. H. Sudborough: "On the tape complexity of deterministic context-free languages", J. ACM., 25, pp.405-414(1978).

(10) I. H. Sudborough: "A note on tape-bounded complexity classes and linear context-free languages", J. ACM., 22, 4, pp.499-500(1975).

(11) C. Wrathall: "Complete sets and the polynomial-time hierarchy", Theoret. Comput. Sci., 3, pp.23-33(1977).